Design Automation for Quantum Architectures

Martin Roetteler Krysta M. Svore Dave Wecker Nathan Wiebe
Microsoft Research, Redmond, WA, USA

***Abstract*—We survey recent strides made towards building a software framework that is capable of compiling quantum algorithms from a high-level description down to physical gates that can be implemented on a fault-tolerant quantum computer. We discuss why compilation and design automation tools such as the ones in our framework are key for tackling the grand challenge of building a scalable quantum computer. We then describe specialized libraries that have been developed using the LIQ$Ui|\rangle$ programming language. This includes reversible circuits for arithmetic as well as new, truly quantum approaches that rely on quantum computer architectures that allow the probabilistic execution of gates, a model that can reduce time and space overheads in some cases. We highlight why these libraries are useful for the implementation of many quantum algorithms. Finally, we survey the tool REVS that facilitate resource efficient compilation of higher-level irreversible programs into lower-level reversible circuits while trying to optimize the memory footprint of the resulting reversible networks. This is motivated by the limited availability of qubits for the foreseeable future.**

I. Introduction

Quantum computing has begun to take off over the last few years. In part, this is thanks to dramatic improvements in hardware and continued improvements in quantum algorithms. Now the technology is perched at the threshold of offering meaningful speedups for problems that are both scientifically and industrially relevant [1], [2], [3], [4].

On the other hand, as we push past this first generation of quantum devices, a host of problems emerge for those who wish to develop software for large scale quantum computers. Even basic questions, such as “how do we program quantum computers?” do not have easy answers. Furthermore, some of our most important quantum algorithms such as linear-systems algorithms and certain classes of quantum simulation algorithms require reversible analogues of arithmetic and elementary functions [5], [6], [7]. Existing quantum circuits for arithmetic require a prohibitive number of quantum bits (qubits) and cannot be automatically generated [8]. Even if such hand crafted networks are provided then verifying and validating them remains a major challenge. These problems need to be solved to realize the promise of quantum computing.

In this survey, we highlight some of the ongoing work towards solving these problems before they prove to be a stumbling block for quantum computing. Researchers at Microsoft Research developed a quantum programming language LIQ$Ui|\rangle$ [9] that enables programmers to translate high-level ideas into concrete quantum circuits and test the resulting networks. In another development, a language called REVS was developed that can compile irreversible classical code into reversible networks—e.g., over the Toffoli gate set—in a way that is cognizant of space and time tradeoffs [10].

Reversible networks have the oft under-appreciated feature that they can be tested efficiently for faults that might be present in a physical implementation, say, on an actual quantum computer [11]. The intended execution can always be predicted, at a fine level of granularity that goes beyond mere end-to-end target functionality. Specifically, we can predict it by partially running the reversible network for a certain number of steps on an input that corresponds to a classical input, i.e., a computational basis state. This might be one advantage that the design of quantum libraries as reversible networks offers over other approaches, e.g., genuine quantum approaches where the identification of faults can be significantly more challenging due to the fact that the computational state is in a superposition, e.g., extended across many basis states, even when the computer is in a debugging mode and probes only basis states.

Another advantage of reversible design is that formal verification techniques can be applied to the problem of ascertaining the correctness of either a given network or a compiler implementation. The former falls in the domain of equivalence checking for circuits, a topic that has been addressed e.g. in [12]. See also [13] for a recent example where correctness of generated circuits was checked by means of BDD based checkers. The latter type of verification falls in the domain of compiler verification where the transformation of the compiler are mathematically proven to be correct, typically with the help of an automatic proof checker. REVER [14], which can be thought of as a restricted version of the REVS compiler, was proven to be correct along these lines by using the proof checker F* [15]. This guarantees that a) any Toffoli network that is generated from an F# functional description by REVER actually matches this functional description as the function computed on the output wires of the network, and b) any clean ancillas that might be needed during the computation are provably returned back into a clean state. Unlike classical computing, ancillas need to be reverted to remove quantum entanglement in quantum computing.

These ideas are then used in a new approach to function synthesis that leverages the decades of progress in irreversible circuit synthesis to automatically generate highly efficient circuits for arithmetic and standard scientific functions that are even more efficient than the best hand-crafted solutions that currently exist. Finally, we discuss new quantum approaches for arithmetic and function synthesis that do not require compilation into classical reversible circuits as an intermediate step [16]. This reveals that even for well understood tasks like arithmetic, quantum computing can provide new ways of thinking about processing data that have no classical analogue.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

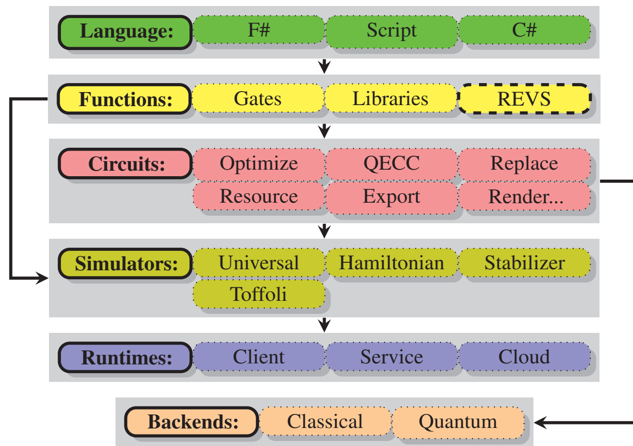

Fig. 1. LIQUi $|\rangle$ architecture overview. Shown are two natural flows, namely the passing of functions to a classical simulator and the passing of emitted circuits to a hardware backend for execution.

II. Quantum programming in LIQUi $|\rangle$

There is no shortage of quantum programming languages. The list includes imperative, C-like languages QCL [17], QLanguage [18], and Scafford/ScaffCC [19], [20], functional languages embedded in Haskell such as Quipper [21], [22], the Quantum I/O Monad [23] and QuaFL/QuiGL [24], [25]. Also, there are several languages for reversible computing based on Janus and various extensions [26], [27], [28]. Furthermore, there are toolkits such as CirKit/RevKit [29] that can assist lower-level functional synthesis from logic descriptions and when put together with a tool flow that can compile from a hardware description language such as Verilog into netlists, can provide high-level to Toffoli gate level compilation.

In the following we describe the approach underlying LIQ$Ui|\rangle$ [9], which is a functional quantum programming language embedded in the .NET host language F# and developed at Microsoft Research. LIQ$Ui|\rangle$ is fundamentally a circuit description language for quantum circuits, however, it also provides support for higher level programming that ultimately will help a programmer to develop quantum computational thinking that is not necessarily centered around circuits but rather on the mathematical functions that are the abstractions of concrete embodiments given by circuits.

In contrast to many of the mentioned approaches, LIQUi $|\rangle$ was designed and architected explicitly with quantum hardware in mind. For instance, one of the underlying design assumptions is that qubits are real entities, have lifetimes and are fundamentally mutable objects, which led to the choice of our basic qubit type. Also, while other languages also come equipped with simulators, the LIQUi $|\rangle$ simulators are highly optimized, taking advantage of many available techniques, including custom memory management, cache coherence analysis, parallelization, "gate growing", and virtualization (running in the cloud). The LIQUi $|\rangle$ simulation environment allows thorough investigation of quantum algorithms under noise, physical device constraints, and simulation.

A. Software architecture

A general overview of the LIQ$Ui|\rangle$ software architecture is shown in Figure 1. While for a more detailed breakdown of the stages we refer to [9], here we provide a brief description of the main features and interplay between stages and highlight particular aspects that are germane to each stage.

- *Language*: The embedding into F# was chosen primarily due to .NET support, object-oriented and functional programming, ease of using reflection and pattern matching which helps with walking complex datastructures, plus a wealth of external libraries. Additions introduced by LIQUi $|\rangle$ to the host language include types Qubit, Ket, Gate, Operation, and Circuit. Notably, besides von Neumann measurements, LIQUi $|\rangle$ also supports also POVMs as a basic gate type.
- *Functions*: The most basic, but at the same time: most important, example of a function is the one of an executable gate. All LIQUi $|\rangle$ gates are user defined functions or class instances that can be extended by the user in many ways. Note that while it is possible to apply introspection to a function (such as asking e.g. whether it corresponds to a unitary operator), it is not mapped to a circuit yet. This allows the freedom to allow many different possible abstract interpretations for any given LIQUi $|\rangle$ function. The Revs compiler, shown as dashed box in Figure 1 can compile classical, irreversible code into LIQUi $|\rangle$ functions that can then be further processed.
- *Circuits*: This level deals with the data structures that most researchers in quantum computing are familiar with: circuits that are composed over an elementary gate set and operate on a certain number of qubits. Many ways to manipulate circuits are possible, including a) further decomposition into lower level gates, b) peep-hole style optimization via rewriting, c) optimizations for trading off size and width, d) resource estimation, e) wrapping with fault-tolerance methods.
- *Simulators*: Simulators that are currently supported include a) a universal simulator that essentially performs the matrix-vector product for a given initial state vector and given unitary corresponding to a circuit. There are various special purpose simulators such as b) a simulator for Clifford circuits based on tracking stabilizer tableaux. Also, there are c) simulators for Hamiltonians, which is particularly useful for applications in quantum chemistry, and d) simulators for reversible circuits, such as Toffoli circuits which is based on tracking basis states as they propagate through the network.
- *Runtimes*: Currently, for development, LIQUi $|\rangle$ supports compilation in environments such as Visual Studio under Windows, as well as full compatibility under Linux/Mono and Mac OS. LIQUi $|\rangle$ is available on *github.com/StationQ/Liquid*. Runtimes supported are client/server versions, as well as an Azure based cloud service. There are several ways in which LIQUi $|\rangle$ code can be executed, e.g., from the command line running the .NET Common Language Runtime, or directly in a Visual Studio interactive session (particularly useful for script files), or in a normal Visual Studio development mode.
- *Backends*: Besides the backend of a classical computer system, many possible quantum computer embodiments can in principle be latched onto via LIQUi $|\rangle$.

B. Hardware design space

Depending on the experimental device design, a quantum computer architecture may allow arbitrary long-range interactions between any two physical qubits or may restrict interactions to occur on spatially neighboring qubits. The former model is represented by a complete graph with an edge between every pair of nodes while the latter model could be represented, for example, by a two-dimensional regular grid. In both cases, we assume concurrent application of gates. Concatenated codes readily support the former abstract concurrent architecture, and may also be designed to support a nearest-neighbor architecture. However, topological codes more naturally support nearest-neighbor restrictions. The surface code tile shown in Figure 2(a) represents one logical qubit encoded in 13 data qubits, with 12 syndrome qubits for error extraction. Data qubits are represented by light nodes and syndrome qubits used to detect errors by dark nodes. Each data qubit may interact with its four neighboring syndrome qubits. In a complete architecture, each logical qubit is encoded in such a tile as shown in Figure 2(b); to enact a two-qubit logical gate, a tile interacts with a neighboring tile through, i.e., lattice surgery [30].

Tiles can also be enlarged to encode several logical qubits, e.g., by introducing so-called defects [31] or displacements [32]. In this case, arbitrary interactions between the qubits within a tile are possible, see e.g. [30], [33], [34]. Considering an intermediate representation to address unitary operations at the level of braids, it is possible to perform two-qubit Clifford operations in a non-local way by braiding in space–time logical qubits that are far away; see [35] for an algorithm that achieves compaction of braids. Here we assume that operations between tiles are brokered via distributed CNOT gates, similar to architectures proposed in [36], [37], [38], [34].

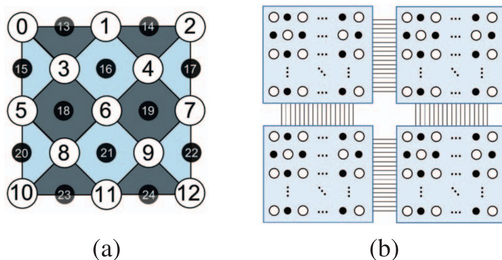

Fig. 2. Surface code quantum architecture: (a) a single qubit tile with data qubits (white) and syndrome qubits (black), shown for the distance-three Surface-25 tile encoding one logical qubit [39]; (b) the overall fault-tolerant architecture assumed in this paper arises from grouping several $n \times n$ surface code tiles, each one holding several logical qubits in a connected network where interactions between neighboring tiles are brokered by CNOTs.

As far as reversible computations are concerned, the number of Toffoli gates used in the implementation of a given permutation is a basic measure of complexity. This measures the circuit *size* as one of the basic metrics we are interested in. Counting Toffolis only is justified from the theory of fault-tolerant quantum computing [40] since the Toffoli gate (and the T gate) has a substantial cost, whereas the cost of so-called Clifford gates, such as CNOT and NOT, can usually be neglected. Another related metric is the overall depth of the circuit, measured usually in the form of T-gate-depth. Implementations of the Toffoli gate over the Clifford+T gate set are known, see e.g. [41] for realizing a Toffoli gate in T-depth 3. The other basic parameter in our design space is the circuit *width*, measured as the maximum number of qubits needed during any point of a quantum circuit, i.e., the maximum number of input qubits, output qubits, and ancilla qubits. Generally, our goal is to trade time for space, i.e., to achieve a reduction in the total number of qubits required. In turn, we are willing to pay a price in terms of a slight increase in the total number of Toffoli gates and in terms of compilation time. Our trade-off is justified by the limited number of qubits available in experimental quantum devices.

III. Reversible programming in Revs

Revs is an embedded language into the .NET language F# and as such inherits some functions and libraries from its host language. Also, the look-and-feel of a typical Revs program is very similar to that of F# programs.

The current implementation of the Revs compiler supports Booleans as basic types only. The core of the language is a simple imperative language over Boolean and array (register) types. The language is further extended with ML-style functional features, namely first-class functions and *let* definitions, and a reversible domain-specific construct *clean*. It should be noted also that Revs was designed to facilitate interoperability with the quantum programming language LIQ$Ui|\rangle$. An example Revs program is shown in Figure 3(a). This example implements a simple carry ripple adder of two n-bit integers. Shown in (b) is one of the possible target intermediate representations, namely LIQ$Ui|\rangle$ code.

	Eager		Bennett			
Name	Bits	Gates	Bits	Gates	Red.(%)	Time
DES	992	85351	2123	62425	53.27	1.720
alu4cl	175	13487	441	8867	60.32	0.101
apex7	111	3299	242	3077	54.13	0.031
b9	127	4444	316	3185	59.81	0.035
c8	80	4791	213	4149	62.44	0.022
comp	97	2606	162	1991	40.12	0.017
cordic	130	3094	262	2363	50.38	0.037
count	71	1304	124	1435	42.74	0.014
cu	36	1395	80	1045	55.00	0.009
dalu	1056	77482	3444	49463	69.34	1.440
decod	27	188	28	365	3.57	0.003
frg2	531	44820	2080	34117	74.47	0.472
lal	86	4051	212	2939	59.43	0.026
mux	45	2841	67	1473	32.84	0.012
rot	325	22407	779	12709	58.28	0.173
sct	64	4509	202	2923	68.32	0.024
term1	181	20744	724	11881	75.00	0.107
toolarge	707	155854	1191	155885	40.64	3.707
ttt2	116	10155	400	8511	71.00	0.062
vda	173	19234	899	19705	80.76	0.140

TABLE I
Space-optimization and compile time results for circuits from the ISCAS, MCNC, and LGSynth benchmarks.

```
let rippleAdder (a:bool []) (b:bool []) =
   let n = Array.length a
   let result =  Array.zeroCreate (n)
   result.[0] <- a.[0] <> b.[0]
   let mutable carry = a.[0] && b.[0]
   result.[1] <- a.[1] <> b.[1] <> carry
   for i in 2 .. n - 1 do
      // compute outgoing carry
      carry <- (a.[i-1]
               && (carry <> b.[i-1]))
               <> (carry && b.[i-1])
      result.[i]  <-  a.[i] <> b.[i]
                      <> carry
   result
```

(a) REVS source program

```
CNOT [ qs.[0]; qs.[10] ]
CNOT [ qs.[5]; qs.[10] ]
CCNOT [ qs.[0]; qs.[5]; qs.[15] ]
CNOT [ qs.[15]; qs.[11] ]
CNOT [ qs.[1]; qs.[11] ]
CNOT [ qs.[6]; qs.[11] ]
CNOT [ qs.[15]; qs.[17] ]
CNOT [ qs.[6]; qs.[17] ]
CCNOT [ qs.[1]; qs.[17]; qs.[16] ]
CCNOT [ qs.[15]; qs.[6]; qs.[16] ]
CNOT [ qs[6]; qs.[17] ]
CNOT [ qs.[15]; qs.[17] ]
CNOT [ qs.[16]; qs.[12] ]
...
```

(b) Segment of the compiled LIQ$Ui|\rangle$ program

Fig. 3. F# program that implements a carry ripple adder using a simple for loop while maintaining a running carry.

We benchmarked the REVS compiler on several combinational circuits, including ISCAS'85, ISCAS'89, MCNC'91, LGSynth'91, LGSynth'92 benchmarks. In Table I the qubit and gate costs for the eager cleanup method and in comparison the corresponding cost for the Bennett cleanup are shown for a subset of the total of 135 circuits we synthesized. Across the entire benchmark, we found an average reduction of 33.48% in terms of the number of qubits, at a price of only a moderate average increase of 10.61% in terms of circuit size. We found improvements in terms of space of as low as 3.57% (instance "decod") and as high as 80.76% (instance "vda").

IV. QUANTUM ARITHMETIC LIBRARIES

We focus on two particular aspects of quantum libraries, namely constant optimization for integer and modular arithmetic, which have direct applications to Shor's factoring algorithm, and space optimization for multiplication and reciprocals of real numbers, which has direct applications to the linear systems quantum algorithm.

A. Constant optimized integer and modular arithmetic

In [11] an implementation of Shor's quantum algorithm to factor n-bit integers using only $2n+2$ qubits. In contrast to previous space-optimized implementations, ours features a purely Toffoli based modular multiplication circuit. The circuit depth and the overall gate count are in $O(n^3)$ and $O(n^3 \log(n))$, respectively. Notably, this approach evades most of the cost overheads originating from rotation synthesis and enables testing and localization of faults in both, the logical level circuit and an actual quantum hardware implementation. We implemented and simulated a Toffoli network for the entire controlled modular multiplication piece of Shor's algorithm in LIQ$Ui|\rangle$, for real-world bit-sizes of up to $8,192$. See Figure 4 for a part of the circuit that implements a modular adder for bit-size 32. Asymptotically, our new (in-place) constant-adder, which is used to construct the modular multiplication circuit, uses only dirty ancilla qubits and features a circuit size and depth in $O(n \log(n))$ and $O(n)$, respectively. Our resource estimates also determine the constants for the scaling of the circuit size.

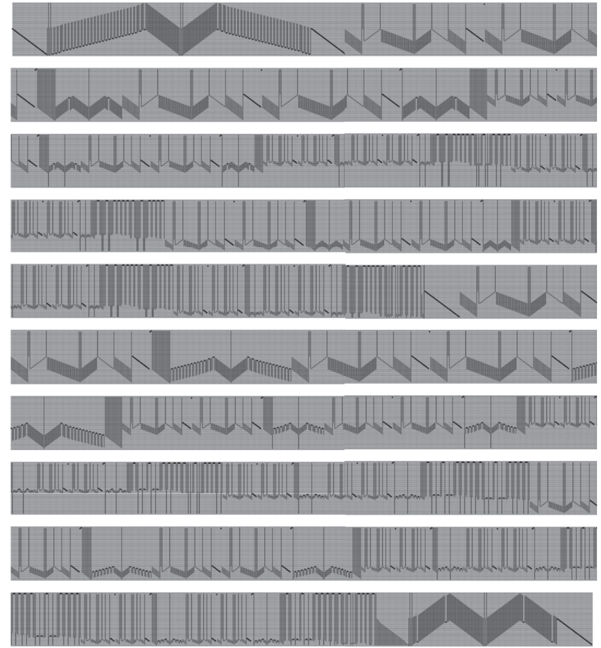

Fig. 4. Toffoli network [11] implementing modular inplace addition $x \mapsto x+a$ of the 32-bit constant $a = 4\,294\,967\,291$ to a 32-bit quantum register. The circuit was rendered in LIQ$Ui|\rangle$, consists of 1813 Toffoli gates, operates on 32 qubits and is part of a larger network implementing the controlled modular multiplication needed in Shor's algorithm. The entire modular multiplication $x \mapsto ax \bmod N$, where N is a 32-bit modulus, requires 66 qubits and a total number of 117 955 Toffoli gates.

B. RUS framework and quantum arithmetic

It is quite common within quantum algorithms to have transformations of the form $|x\rangle\,|\psi\rangle \rightarrow |x\rangle\, e^{if(x)Z}\,|\psi\rangle$, for some function f that maps $\mathbb{B}_n \rightarrow \mathbb{R}$. This form of transformation occurs in algorithms such as the linear systems algorithm [5] and also is ubiquitous in quantum machine learning algorithms [42], [43]. If we were synthesizing the function f through a reversible circuit we would be in essence approximating $\mathbb{F} \approx \mathbb{F}'$ where $\mathbb{F}' : \mathbb{B}_n \mapsto \mathbb{B}_m$ where m is the number of bits of precision for the output. Then conditioned on

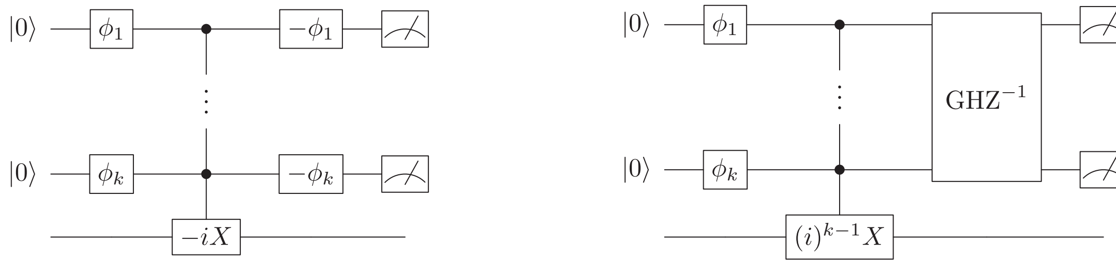

Fig. 5. GB (left) and PAR circuit (right) with k inputs, where GHZ^{-1} is the inverse GHZ measurement. Success occurs if all measurements read 0. PAR implements an approximate multiplication on success and GB an approximate square-multiplier.

the m-bit output the required quantum transformation $e^{if(x)Z}$ can be synthesized using m controlled single-qubit rotations. Since these m qubits are only being used in an intermediate step in the algorithm, one can try to avoid them.

To achieve this goal, we need to introduce non-linearities. One approach to generating non-linearities exploits measurement, which is the only non-linear operation in quantum mechanics. We use two circuits shown in Figure 5 to generate these transformations [16]. These circuits take its inputs to be the rotation angles that specify a given single-qubit rotation. They then aim to output a rotation that rotates a qubit through an angle that is approximately the product of the input rotation angles (PAR circuit), or product of their squares (GB circuit). These circuits achieve this, with high probability, given that their inputs are small. Furthermore, they can be corrected and the operation can be tried anew if and when they fail. This "repeat-until-success" (RUS) feature is key to their efficiency.

A feature of this approach to constructing an approximate multiplier is that two small numbers can be multiplied in this fashion using only 2 ancillary qubits independent of the precision of the inputs. This is because we use an analog degree of freedom, rather than a digital representation, to store the result. There is, of course, no free lunch. Higher precision necessitates more error correction, but this only incurrs a logarithmic space overhead and often will be irrelevant given the exacting precision already required of quantum computers by certain algorithms.

Larger numbers can be multiplied using these circuits using ideas from split operator formulas. In particular, $\text{PAR}(a, b) = ab + O(\max(a, b)^4)$ and hence for any integer $r > 0$, $r\text{PAR}(a/\sqrt{r}, b/\sqrt{r}) = ab + O(\max(a, b)^4/r)$. Here multiplication by r is implemented by repeating the circuit r times. Hence by taking $r \to \infty$ the error can be made arbitrarily small. Similar tricks, and an algorithm called oblivious amplitude amplification, can be used to make the success probability asymptotically independent of the input rotation angles [16].

This simple version of the algorithm can be optimized by constructing higher-order multiplication formulas that use GB and PAR to construct a higher-order Taylor series approximation to the product. As an example $ab = \text{PAR}(a, b, \frac{\pi}{4} - \text{GB}(\gamma_2, a) - \text{GB}(b, \phi_2)) + O(\max(a, b)^6)$ for $\gamma_2 = \arcsin(1/\sqrt{6})$. We refer to these two lowest order formulas as M_4 and M_6 respectively. By iteratively increasing the order of accuracy of the multiplier, r can be made to grow sub-polynomially with the reciprocal of the error tolerance [16].

We examine the performance of these multipliers in the table below relative to carry-ripple adders and Table-lookup for computing 0.5^2 at $n = 8, 16$ bits of precision in the output [16].

	$n = 8$		$n = 16$	
Multiplier	T-count	qubits	T-count	qubits
Carry-ripple	2.80E+03	16	1.06E+04	32
Table-lookup	3.98E+09	2	1.13E+13	2
M_4	4.64E+04	4	3.00E+07	4
M_6	3.82E+03	4	5.21E+05	5

We see that while the Carry-Ripple multiplier requires more qubits as the precision increases, our multiplier M_4 does not. In fact, the only reason why M_6 needs an extra qubit at $n = 16$ is because that is the first time where $r > 1$ is needed. This forces us to apply the entire circuit as an RUS-circuit and which requires us to use an additional qubit. No additional ancillary qubits will be needed as we increase n beyond 16 for M_6. The number of T–gates is intermediate between Table-lookup (which is untenable here) and the Carry-Ripple multiplier. This demonstrates the space efficiency of our approach and shows that genuinely quantum methods for arithmetic are possible that can reach better tradeoffs between resources than classical strategies can.

As a final note, from the perspective of our synthesis algorithms there is nothing special about multiplication. We can synthesize any function that is analytic on a compact domain using the Taylor series approach. Furthermore, this approach can also be used to implement a Chebyshev approximation to a function or even approximate an arbitrary function as a sum of piecewise constant functions [16]. This flexibility shows that the basic ideas behind this approach are much more general than it may seem at first glance. The ability to compute arbitrary functions in the phase of a qubit provides more than just the first genuinely quantum algorithms for arithmetic: it reveals a new type of "quantum numerical analysis".

V. Conclusion

As we progress from today's demonstrations of quantum technologies to tomorrows large scale quantum computers, we

need to have design automation and programming tools in place. Without such tools, the challenges of designing and testing quantum circuits to implement meaningful algorithms by hand will be too daunting.

Finally, we still need to think broadly about how to approach arithmetic or programming in quantum computing. While present-day approaches that mimic solutions that have been successful for classical computing provide adequate solutions for now, it is likely that the solutions of tomorrow may fundamentally differ from anything we can currently imagine.

REFERENCES

[1] R. Barends, J. Kelly, A. Megrant, A. Veitia, D. Sank, E. Jeffrey, T. White, J. Mutus, A. Fowler, B. Campbell *et al.*, "Superconducting quantum circuits at the surface code threshold for fault tolerance," *Nature*, vol. 508, no. 7497, pp. 500–503, 2014.

[2] A. W. Cross, G. Smith, and J. A. Smolin, "Quantum learning robust against noise," *Physical Review A*, vol. 92, no. 1, p. 012327, 2015.

[3] P. O'Malley, R. Babbush, I. Kivlichan, J. Romero, J. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding *et al.*, "Scalable quantum simulation of molecular energies," *Physical Review X*, vol. 6, no. 3, p. 031007, 2016.

[4] M. Benedetti, J. Realpe-Gómez, R. Biswas, and A. Perdomo-Ortiz, "Estimation of effective temperatures in quantum annealers for sampling applications: A case study with possible applications in deep learning," *Physical Review A*, vol. 94, no. 2, p. 022308, 2016.

[5] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations," *Physical review letters*, vol. 103, no. 15, p. 150502, 2009.

[6] R. Babbush, D. W. Berry, I. D. Kivlichan, A. Y. Wei, P. J. Love, and A. Aspuru-Guzik, "Exponentially more precise quantum simulation of fermions II: Quantum chemistry in the CI matrix representation," *arXiv preprint arXiv:1506.01029*, 2015.

[7] I. D. Kivlichan, N. Wiebe, R. Babbush, and A. Aspuru-Guzik, "Bounding the costs of quantum simulation of many-body physics in real space," *arXiv preprint arXiv:1608.05696*, 2016.

[8] M. K. Bhaskar, S. Hadfield, A. Papageorgiou, and I. Petras, "Quantum algorithms and circuits for scientific computing," *arXiv:1511.08253*, 2015.

[9] D. Wecker and K. Svore, "LIQ*ui*|⟩: a software design architecture and domain-specific language for quantum computing," arXiv.org preprint arXiv:1402.4467.

[10] A. Parent, M. Roetteler, and K. M. Svore, "Reversible circuit compilation with space constraints," *arXiv*, vol. 1510.00377, 2015.

[11] T. Häner, M. Roetteler, and K. M. Svore, "Factoring using 2n+2 qubits with toffoli based modular multiplication," *arXiv*, vol. 1611.07995, 2016.

[12] J. P. H. George F. Viamontes, Igor L. Markov, "Checking equivalence of quantum circuits and states," in *Proceedings ICCAD'07*, 2007, pp. 69–74.

[13] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Design automation and design space exploration for quantum computers," in *Proceedings DATE'17*, 2017, accepted for publication.

[14] M. Amy, M. Roetteler, and K. M. Svore, "Verified compilation of space-efficient reversible circuits," *arXiv*, vol. 1603.01635, software available at *github.com/msr-quarc/ReVerC*.

[15] N. Swamy, C. Hricu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *POPL'16*, 2016, pp. 256–270.

[16] N. Wiebe and M. Roetteler, "Quantum arithmetic and numerical analysis using repeat-until-success circuits," *Quantum Information and Communication*, vol. 16, pp. 134–178, 2016.

[17] B. Oemer, "Classical concepts in quantum programming," *International Journal of Theoretical Physics*, vol. 44, no. 7, pp. 943–955, 2005.

[18] S. Bettelli, T. Calarco, and L. Serafini, "Toward an architecture for quantum programming." *European Physics D*, vol. 25, no. 2, pp. 181–200, 2003.

[19] J. Heckey, S. Patil, A. Javadi Abhari, A. Holmes, D. Kudrow, K. R. Brown, D. Franklin, F. T. Chong, and M. Martonosi, "Compiler management of communication and parallelism for quantum computation," in *ASPLOS'15*, 2015, pp. 445–456.

[20] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "ScaffCC: Scalable compilation and analysis of quantum programs," *Parallel Computing*, vol. 45, pp. 2–17, 2015.

[21] A. S. Green, P. LeFanu Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "An introduction to quantum programming in Quipper," in *Proc. Reversible Computation (RC'13)*. ACM, 2013.

[22] ——, "Quipper: a scalable quantum programming language," in *Proc. Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, 2013.

[23] T. Altenkirch and A. S. Green, "The quantum IO monad," in *Semantic Techniques in Quantum Computation*, S. Gay and I. Mackie, Ed. Cambridge University Press, 2009, pp. 173–205.

[24] A. Lapets, M. da Silva, M. Thome, A. Adler, J. Beal, and M. Roetteler, "QuaFL: A typed DSL for quantum programming," in *Proceedings of Workshop on Functional Programming Concepts in Domain-Specific Languages (FPCDSL'13)*. ACM Press, 2013, pp. 19–26.

[25] A. Lapets and M. Roetteler, "Abstract resource cost derivations for logical quantum circuit descriptions," in *Proceedings of Workshop on Functional Programming Concepts in Domain-Specific Languages (FPCDSL'13)*. ACM Press, 2013, pp. 35–42.

[26] T. Yokoyama and R. Glück, "A reversible programming language and its invertible self-interpreter," in *Proc. Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'07)*. ACM, 2007, pp. 144–153.

[27] M. K. Thomsen, "A functional language for describing reversible logic," in *Proc. Forum on Specification and Design Languages (FDL'12)*. IEEE, 2012, pp. 135–142.

[28] K. S. Perumalla, *Introduction to Reversible Computing*. CRC Press, 2014.

[29] M. Soeken, S. Frehse, R. Wille, and R. Drechsler, "RevKit: A toolkit for reversible circuit design," *Multiple-Valued Logic and Soft Computing*, vol. 18, no. 1, pp. 55–65, 2012.

[30] C. Horsman, A. G. Fowler, S. Devitt, and R. Van Meter, "Surface code quantum computing by lattice surgery," *New J. Phys*, vol. 14, p. 123011, 2012.

[31] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," *Phys. Rev. A*, vol. 86, p. 032324, 2012, arXiv:1208.0928.

[32] M. B. Hastings and A. Geller, "Reduced space-time and time costs using dislocation codes and arbitrary ancillas," *Quantum Information and Computation*, vol. 15, pp. 962–986, 2015.

[33] I. Polian and A. G. Fowler, "Design automation challenges for scalable quantum architectures," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, 2015, pp. 61:1–61:6.

[34] S. C. B. Ying Li, "Hierarchical surface code for network quantum computing with modules of arbitrary size," 2015, arxiv preprint arXiv:1509.07796.

[35] A. Paetznick and A. G. Fowler, "Quantum circuit optimization by topological compaction in the surface code," 2013, arxiv preprint arXiv:1304.2807.

[36] T. Metodi, A. Faruque, and F. Chong, *Quantum computing for computer architects*, 2nd ed., ser. Series: Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[37] R. van Meter, K. Nemoto, W. J. Munro, and K. M. Itoh, "Distributed arithmetic on a quantum multicomputer," in *33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA*, 2006, pp. 354–365.

[38] R. van Meter, *Quantum Networking*. John Wiley & Sons, 2014.

[39] Y. Tomita and K. M. Svore, "Low-distance surface codes under realistic quantum noise," *Phys. Rev. A*, vol. 90, p. 062320, 2014.

[40] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge, UK: Cambridge University Press, 2000.

[41] M. Amy, D. Maslov, M. Mosca, and M. Roetteler, "A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 6, pp. 818–830, June 2013.

[42] S. Lloyd, S. Garnerone, and P. Zanardi, "Quantum algorithms for topological and geometric analysis of data," *Nature communications*, vol. 7, 2016.

[43] N. Wiebe, A. Kapoor, and K. M. Svore, "Quantum deep learning," *Quantum Information and Computation*, vol. 16, pp. 0541–0587, 2016.